Little worlds with big implications: Software as miniature

Luke Church
Computer Laboratory
Cambridge University
luke@church.name

Mariana Marasoiu
Computer Laboratory
Cambridge University
mariana.marasoiu@cl.cam.ac.uk

Abstract

Programming has a feel to it. Sometimes it feels light and airy, other times tangled and bureaucratic. The PPIG community, and others, have been trying to get at what's going on here for years. In this paper we suggest one more route in. Tuan's lens of Dominance and Affection describes how owners regard their pets. The same ideas extend to the doll's house, and often extend to the software and tools used to build it. By comparing what happens in a doll's house and a software program we make some suggestions as to what the implications are for the experience that programmers have and how an alternative programming might look like.

Keywords: ontology, models, objects

Introduction

That software needs to be "usable" is no longer a claim that needs to be justified, even when the users of the software are themselves developers. It is inconceivable that one would build developer tools intended for general usage without considering usability. But whilst 'usability' or 'productivity' have become common ways of marketing developer products or prototypes, we suggest that they are insufficient to describe the experience of programmers in the wild.

This experience is quite an emotional one. Developers engage in impassioned debates over which editor to write their code in, EMACS versus vi being the canonical example (Smith, 2001); whether indentation should be done using tab characters or spaces (Zawinski, 2000); or how long lines should be. In these "holy wars" (Cohen, 1981) advocacy for one position or another is based on instinct and personal preference, presented as usability arguments, even when there is little or no data demonstrating a significant difference between the options. To take just one example, Peterson et al. (2019) found no correlation between time spent reading the line and line length, and yet findings such as these do not dissuade long-held beliefs that line length matters, or more often, that a certain line length is better than another.

What's happening here? Arguments about personal preference are being hotly presented as arguments about usability, but there is little or no evidence of any effect on performance. One view might be that these arguments are just identity politics, that by taking and defending particular positions, developers are establishing themselves within a group. Another view, the one we shall take here, is to try and understand the phenomena that the developers are experiencing, that is, to understand the programmer's "lived experience" (Husserl, 1970). Less has been said from this perspective, and it is a remarkably difficult route to follow; for a start, the phenomenology applies in a complex couple to both the place where development happens (the desk, the laptop, the keyboard, the IDE, the tab layout) and the thing being created (the program, the language, the product). Here we'll attempt to make a small step in this direction by looking at the relationship of the programmer with their program, and what the implications of that might be for the programmer's engagement with the wider world through the program. In order to do this we'll refer to the work of the Chinese-American humanistic geographer 段義孚, and his essay on the creation of pets (Tuan, 1984).

Dominance and affection: the making of software?

Tuan posited that pets are made by humans where the desire for dominance intersects with affection for the object. He explores this concept with a wide array of examples, both present and historical, including manicured gardens, zoos, doll's houses, bonsai, and specially bred pet fish and dogs. He argues that the pet is an attempt to exert control over nature in the context of play or entertainment, instead of work and conquest.

From our own experience as programmers, and from having seen and worked with other programmers, both components that Tuan mentions - dominance ("I need to get this code to do what I want it to do") and affection ("I've enjoyed writing this neat piece of code today") are part of how we feel towards the software that we build. The word "pet" alongside "last weekend's coding project that is now running on a Raspberry Pi on my desk" doesn't seem completely ill-fitting.

Is there anything then that we can learn from Tuan's analysis of pet-making when it comes to programming?

The programmer's bonsai

Tuan points out that pets are idealised, tamed and reductive versions of real things, and thus pet-making is the process of controlling and directing what the *pet* should look like and behave like. Gardeners likewise take nature and coerce it into a shape perceived as pleasurable by the landowner. Dogs are bred to emphasise traits that will make them more attractive to the owners, and also spayed and neutered to reduce undesired behaviours. Tigers in some zoo exhibits used to be fed every day at a scheduled time in order for the human visitors to enjoy the spectacle, even if in the wild, they might only have one good meal a week.

In software we notice an aesthetic desire for an artificial clarity within the program being written at the expense of what the program was supposed to be doing. We've mentioned above some examples of this, indentation, line length, but there are many others, often discussed under the umbrella of "coding style". There's a desire to build clean and tidy software rather than useful software. This tension becomes more serious when it moves away from the niceties of code, and applies to what the code does. We have talked in the past (Blackwell et al., 2008) about the moral hazard of the essentialism embedded within Computational Thinking, but what we never explained was why it was so persuasive to so many; perhaps it's precisely because the cleaned versions of the world it claimed captured the essence of the world, were in fact programmer's pets.

Another kind of pet can be found when we look at the tools that the programmer uses. With the configurability of the developer environments and the diversity of tools available, programmers are able to particularise their coding setup, and adapt it, but also adapt to it. Over time, it becomes familiar. They become hyper-sensitised to the tools they use, in the same way that people become attuned to their home.

With developer tooling changing so often in small ways that programmers cannot control, what might a programmer who has set up their environment "just so" experience when the inevitable automatic update happens? It's not a loss of productivity, it's frustration, disorientation and in significant cases, grief at the loss of the pet they have lovingly constructed.

Upgrading software, then, is telling people not to worry. Here is a better dog.

Mistaking miniatures for the real thing

Doll's houses are toy houses made in miniature. Idealised, reduced, fully controllable versions of the real house. Places where children learn and adults reminisce of a simpler time. They're in a different category from the house itself - we cannot say that the doll's house is a representative model of what happens in a real house, but we make this model-representation fallacy all the time in our tidy software. The problem is that whilst the chair in the doll's house reminds you every time you look at it that you can't sit on it, our programs are so abstract that they don't remind us in the same way.

Reflecting on the miniaturisation process that Tuan discusses, Scott (1998) points out that the same kind of desire for control also applies to large scale bureaucracies. The miniaturisation then is a pursuit of an aesthetic - idealised, reduced, controllable, legible versions of the world - and not the pursuit of utility within the real world. When this is coupled with the dominant power of the modern state, tragedies can and have happened.

In software, the power the programmer seeks to wield over their miniature world is not so very different, and is often in the same pursuit of the aesthetic of high modernity - order, clarity, legibility, capturing the essence. Indeed our programming languages make things that don't fit into this clarity unsayable in a Wittgensteinian manner.

This makes the underlying claim behind constructivism in Computer Science - that by building software for something you've understood the thing - a risky position. You wouldn't claim that having built a dolls' house, you can go on to build a real house.

Design for pets

So what to do? Firstly, if software engineers treat what they are creating more like a pet than like the products coming out of a factory, then they need implements that resemble a bonsai toolkit rather than tractor attachments.

Secondly, we need to let go of usability as a determining logic of the selection of programming languages and tools. It may once have been, but study after study shows no-significance now, and yet that doesn't stop the choices really mattering to programmers. We need to understand why.

Thirdly, and perhaps most crucially, we need to develop an aesthetic sense for software that works with humility in the world rather than dominating it. More the old woodland than the model garden. The lessons of humility and living with nature apply to software too.

References

Blackwell, A. F., Church, L., & Green, T. (2008). The Abstract is "an Enemy": Alternative Perspectives to Computational Thinking. *Proceedings PPIG 2008, the 20th Annual Workshop of the Psychology of Programming Interest Group*. http://www.ppig.org/papers/20th-blackwell.pdf

Cohen. (1981). On Holy Wars and a Plea for Peace. *Computer*, *14*, 48–54.

Husserl, E. (1970). *The Crisis of European Sciences and Transcendental Phenomenology: An Introduction to Phenomenological Philosophy*. Northwestern University Press.

Peterson, C. S., Abid, N. J., Bryant, C. A., Maletic, J. I., & Sharif, B. (2019). Factors influencing dwell time during source code reading: a large-scale replication experiment. *Proceedings of the 11th ACM Symposium on Eye Tracking Research & Applications*, 1–4.

Scott, J. C. (1998). *Seeing like a state*. Yale University Press.

Smith, J. T. (2001, December 4). EMACS vs. vi: The endless geek “holy war.” *Linux.com*. https://www.linux.com/news/emacs-vs-vi-endless-geek-holy-war/

Tuan, Y.-F. (1984). *Dominance and Affection: The Making of Pets*. Yale University Press.

Zawinski, J. (2000). Tabs versus Spaces. *Jwz.org*. https://www.jwz.org/doc/tabs-vs-spaces.html